Open in app ↗ Sign up Sign in

Medium Search Write

Reverse Engineering Game Memory: How to hack any game using Cheat Engine & P/Invoke in .NET

Avtandil Ushikishvili Follow 9 min read · Feb 16, 2025

8 1

Wile E. Coyote could have been a genius hacker and saved us from CHEATERS

Introduction

Reverse engineering game memory has been a long-time practice among security researchers, ethical hackers, and gaming enthusiasts looking to modify in-game values. One of the most popular tools for this is **Cheat Engine**, which allows users to search for and modify memory values in a running game process.

However, this technique is not limited to Cheat Engine alone. With **P/Invoke** (Platform Invocation Services) in .NET, we can interact with Windows API functions from `kernel32.dll` to read and write process memory. In this article, we will explore how game memory modification works, how Cheat Engine helps in locating key memory addresses, and how to programmatically manipulate those values using C#.

In this article, we will explore:

- How **memory addresses** work in game hacking.
- How to **identify and modify values** using **Cheat Engine**.
- How to programmatically alter values using **P/Invoke in C#**.
- A practical **example modifying health in AssaultCube**.

***Disclaimer**: This article is for educational purposes only. Modifying game memory can violate terms of service in online games and lead to bans. Use these techniques responsibly.*

Understanding Game Memory and Cheat Engine

How Game Memory Works

Every running game stores variables such as **player health, ammo, or coordinates** in memory. These values are stored at specific addresses, which can be accessed and modified — if we have the right permissions.

However, memory addresses are often **dynamic** (changing every time the game starts). This is where Cheat Engine helps by searching for values, tracing pointers, and allowing modifications.

Understanding Memory Addresses in Game Hacking

Memory addresses are the foundation of game hacking. Every variable in a game — whether it's **health, ammo, money, or coordinates** — is stored at a **specific memory address** while the game is running. If we can find these addresses, we can manipulate them to alter in-game values, giving us control over certain mechanics like **infinite health, unlimited ammo, or increased speed**.

However, modifying memory isn't always straightforward. Many games use **dynamic memory allocation**, making the process more complex, requiring deeper knowledge of pointers and memory scanning techniques.

What is a Memory Address?

A **memory address** is a **unique identifier** for a location in RAM where a value is stored. Think of it like a **house address** — every variable has its own **unique location** in memory where its data is kept.

For example, in a game, your **player's health** might be stored at:

```
0x006F4A60
```

This means that whenever the game **needs to check or update your health**, it **reads from and writes to** this location in memory.

However, modern games often **do not use fixed addresses** for important variables. Instead, they allocate memory **dynamically**, meaning that each time you start the game, the address where your health is stored might change. This requires us to use **pointer scanning** to track down the **real** location of the data.

How Games Store Values in Memory

Games use different **data types** to store variables in memory:

- **Integers (int, short, long)** → Used for whole numbers like **health, ammo, score, or money**. In Cheat Engine, these are typically **4 Bytes**.
- **Floating-point numbers (float, double)** → Used for precise values like **speed, player coordinates, or weapon spread.**
- **Boolean (bool)** → Used for states (e.g., `1 = isAlive`, `0 = isDead`).
- **Strings (char[])** → Used for **player names, chat messages, or dialogue texts.**
- **Pointers** → Used for **dynamic memory management**, allowing the game to reference memory addresses indirectly rather than storing values at fixed locations.

Static vs. Dynamic Memory Addresses

Memory addresses in games can be **static** or **dynamic**. Understanding the difference is crucial for modifying values effectively.

Static Address:

- A **static address does not change** between game sessions.
- If your health is **always** stored at `0x006F4A60`, it is a **static address.**
- Older or simpler games (such as single-player indie games) often have **static** memory addresses.

💡 **Example:**
If a game developer hardcodes health at `0x006F4A60`, you can modify it directly every time the game runs.

Dynamic Address:

- **Most modern games use dynamic memory allocation**, meaning values are assigned new memory addresses **each time** the game starts.
- If you restart the game, the value at `0x006F4A60` might now store something else, while your **health might be at a completely different address**, such as `0x00AF6D20`.
- This is why we use **pointers** to track dynamic memory locations.

💡 **Example:**
In a modern FPS game, the player's health might be dynamically allocated **each time you respawn.** If you only store `0x006F4A60` from one game session, it won't work when you restart.

To reliably find the **real memory address**, we need to use **pointer scanning.**

How Cheat Engine Helps in Memory Scanning

Finding Static Addresses (Direct Search)

1. **Open Cheat Engine** and attach it to the game process.
2. Search for the **current health value** (e.g., `100` if your health is 100).
3. **Take damage**, then scan for the **new health value**.
4. Repeat this process **until only one or a few addresses remain**.
5. If the address remains the same after restarting, it is a **static address**.

If the address changes after restarting, the game is using **dynamic memory allocation**, and we need to find **pointers**.

Handling Dynamic Memory Addresses (Pointer Scanning)

1. After finding the **temporary memory address** where health is stored, right-click on it in Cheat Engine and select "**Find out what accesses this address.**"
2. Take damage or heal in the game. Cheat Engine will show a **list of instructions that modify the value**.
3. Click on one of these instructions and **view the pointer details**.
4. Instead of modifying the **temporary memory address**, follow the pointer chain to the **base address**.
5. Use **pointer scanning** to track where the **actual** data is stored.

How Cheat Engine Helps Identify Memory Values

Cheat Engine works by scanning **entire memory regions** within a process and identifying locations where specific values are stored. It uses several techniques to refine the results:

Cheat Engine Memory Scanning Features

- **Exact Value Search** → If you know the exact value (e.g., health = `100`), you can search for it directly.
- **Unknown Initial Value** → If you're unsure, you can scan for any **changing values.**
- **Decreasing / Increasing Values** → Helps track **health loss, ammo consumption, or speed changes.**
- **Pointer Scan** → Finds **multi-level pointers** that lead to dynamic memory addresses.
- **Memory Region Analysis** → Allows exploration of how the game manages different types of memory (e.g., heap vs. stack).

Example: How Cheat Engine Helps Find Game Values

Let's say we want to modify a game's **health.**

Get Avtandil Ushikishvili's stories in your inbox

Join Medium for free to get updates from this writer.

Enter your email

Subscribe

Find the Value Manually

- Start the game.

- Open Cheat Engine and attach it to the process.
- Search for the **current health** (e.g., `100`).
- Take damage and **search again** for the updated value.
- Repeat until you **find the correct address.**

Determine if the Address is Static or Dynamic

- Restart the game and check if the address still holds the **correct** health value.
- If it changes, the address is **dynamic**, and we need to find a **pointer.**

Find the Pointer (Base Address)

- Right-click on the address in Cheat Engine and select "**Find out what writes to this address.**"
- Identify the **game instruction** modifying the value.
- Use **pointer scanning** to locate the **true** memory address.

Using P/Invoke in .NET to Modify Game Memory

Now that we've learned how to find memory addresses using Cheat Engine, the next step is modifying those values **programmatically.** This is where **P/Invoke** comes in.

What is P/Invoke?

P/Invoke, short for **Platform Invocation Services**, is a feature in .NET that allows **C# code to call native Windows API functions.** Since Windows API

functions are written in **C and C++**, they operate outside the managed .NET environment. P/Invoke acts as a bridge that allows us to **interact with system-level functions** directly from C#.

Why Do We Need P/Invoke for Game Memory Hacking?

Most games **do not expose** their internal data to external programs. Unlike applications that offer APIs, **games do not provide direct access** to their memory structures. Since game data is stored in **protected memory**, we need to bypass these restrictions to **read or modify in-game values.**

Windows provides **low-level system functions** in `kernel32.dll` that allow us to:

- **Open a running game process.**
- **Read memory values from the game's memory space.**
- **Write new values to specific memory locations.**
- **Close the process after modifying values.**

With P/Invoke, we can **programmatically change game variables**, just like we did manually in Cheat Engine.

Key Windows API Functions Used in Game Hacking

We'll be using **four main Windows API functions** from `kernel32.dll` to interact with game memory:

`OpenProcess` – Gain Access to a Process

This function allows us to **open another process** (the game) so we can read and modify its memory.

```
[DllImport("kernel32.dll")]
public static extern nint OpenProcess(int dwDesiredAccess, bool bInheritHandle,
```

🔹 **Usage:** This is the first step in memory hacking. Without access to the process, we cannot modify its memory.

`ReadProcessMemory` – Read Memory from Another Process

This function allows us to **read** the current value stored at a given memory address in another process.

```
[DllImport("kernel32.dll", SetLastError = true)]
public static extern bool ReadProcessMemory(nint hProcess, nint lpBaseAddress, b
```

🔹 **Usage:** We use this function to check **what value is currently stored in memory**, ensuring that we have found the correct memory address.

`WriteProcessMemory` – Modify Memory Values

This function **writes new values** to a specific memory address in another process.

```
[DllImport("kernel32.dll", SetLastError = true)]
public static extern bool WriteProcessMemory(nint hProcess, nint lpBaseAddress,
```

🔹 **Usage:** After identifying the correct memory address, we use this function to **modify** the stored value.

`CloseHandle` – Close the Process Handle

This function **closes the handle** to the game process, ensuring we properly release system resources.

```
[DllImport("kernel32.dll")]
public static extern bool CloseHandle(nint hObject);
```

🔹 **Usage:** Always **close the process handle** after modifying memory to avoid unnecessary system resource consumption.

Example: Using P/Invoke to Modify Health in AssaultCube

We will now use **P/Invoke** to programmatically modify player health in the **AssaultCube** game.

Objective:

- Open the game process.
- Read the **current health value** from memory.
- Modify the health value to **999**.
- Close the process handle after modification.

Demo:

```
using System.Diagnostics;
using System.Runtime.InteropServices;

namespace GameMemoryHacking;
class Program
{
    [DllImport("kernel32.dll")]
    public static extern nint OpenProcess(int dwDesiredAccess, bool bInheritHand

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern bool ReadProcessMemory(nint hProcess, nint lpBaseAddres

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern bool WriteProcessMemory(nint hProcess, nint lpBaseAddre

    [DllImport("kernel32.dll")]
    public static extern bool CloseHandle(nint hObject);

    const int PROCESS_ALL_ACCESS = 0x1F0FFF;

    static void Main()
    {
        string gameProcessName = "ac_client"; // Name of the AssaultCube process
        var healthAddress = (nint)0x006F4A60; // Memory address of health

        var processes = Process.GetProcessesByName(gameProcessName);
        if (processes.Length == 0)
        {
            Console.WriteLine("Game process not found.");
            return;
        }

        var gameProcess = processes[0];
        var hProcess = OpenProcess(PROCESS_ALL_ACCESS, false, gameProcess.Id);

        if (hProcess == nint.Zero)
        {
            Console.WriteLine("Failed to open process.");
            return;
        }

        var buffer = new byte[4];
        ReadProcessMemory(hProcess, healthAddress, buffer, (uint)buffer.Length,
        var currentHealth = BitConverter.ToInt32(buffer, 0);
        Console.WriteLine($"Current Health: {currentHealth}");

        var newHealth = 999;
        var newHealthBytes = BitConverter.GetBytes(newHealth);
```

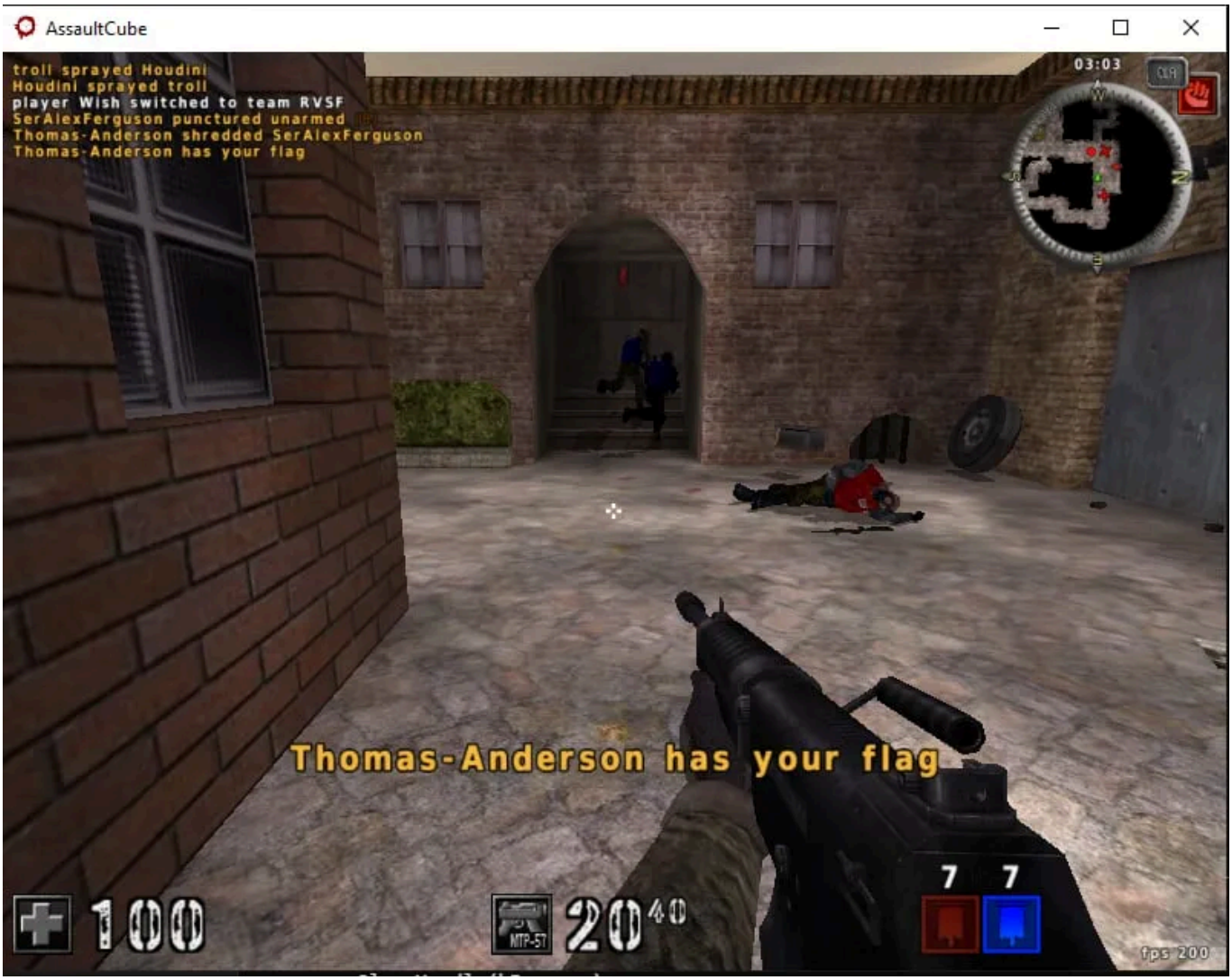
```
            WriteProcessMemory(hProcess, healthAddress, newHealthBytes, (uint)newHea
            Console.WriteLine("Health modified successfully!");

            CloseHandle(hProcess);
        }
    }
```

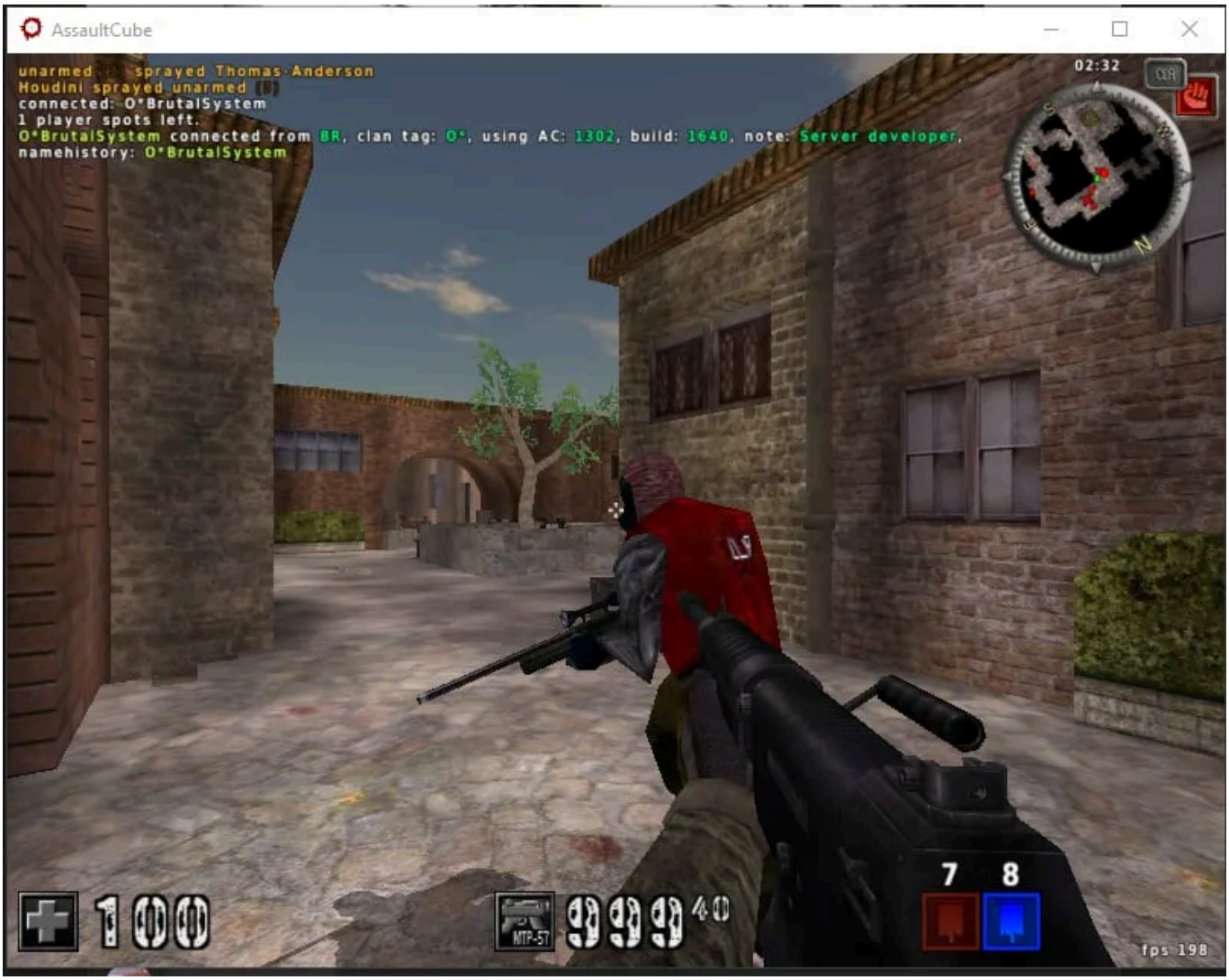
Step-by-Step Breakdown

1. **Attach to the game process** (`ac_client.exe` for AssaultCube).
2. **Read the current health value** stored in memory.
3. **Modify the health value** from whatever it was to `999` .
4. **Write the new value back** into memory.
5. **Close the process handle** to clean up resources.

Before:


AssaultCube
troll sprayed Houdini
Houdini sprayed troll
player Wish switched to team RVSF
SerAlexFerguson punctured unarmed
Thomas-Anderson shredded SerAlexFerguson
Thomas-Anderson has your flag
03:03
Thomas-Anderson has your flag
100
MTP-57
20 40
7 7
fps 200

After:



P.S This was done in multiplayer match !

Why This Works

- **Cheat Engine helped us locate the correct memory address** (`0x006F4A60`) for health.
- **P/Invoke allows us to access and modify this address programmatically.**
- **We successfully alter the game state dynamically.**

This is **one of the simplest** examples of game memory hacking using **P/Invoke**, but the same principles apply to **ammo, money, player speed, or even AI behavior.**

There are many other ways to manipulate game memory, including enabling wallhacks, aimbots, speed hacks, and more. The feasibility and methods of hacking a game depend heavily on various factors, such as:

- **The Game Engine:** Different engines (Unity, Unreal Engine, Source, CryEngine, etc.) handle memory management differently. Some use encrypted values, while others rely on structured object hierarchies.
- **How the Game Stores Variables:** Some games use **static memory addresses**, making them easier to modify, while others rely on **dynamic memory allocation**, requiring pointer resolution.
- **Protection Mechanisms:** Many modern games implement **anti-cheat systems** (e.g., BattlEye, Easy Anti-Cheat, VAC), which actively detect and prevent memory manipulation.
- **Rendering API Used:** Whether the game is built on **OpenGL, DirectX, Vulkan, or another rendering API** affects how graphics-based exploits like wallhacks or ESP (Extra Sensory Perception) work.
- **Code Obfuscation & Virtualization:** Some developers use techniques like **code obfuscation, function hooking prevention, or memory encryption** to make hacking more difficult.

While Cheat Engine and P/Invoke allow direct memory manipulation, **more advanced techniques** involve modifying the game's logic at a **deeper level**. The complexity of these techniques depends on how **well the game protects itself** against such modifications.

Happy hacking. 🚀 🎮

C Sharp Programming | Dotnet | Cheat Engine | Ethical Hacking | Cheating

Written by Avtandil Ushikishvili

24 followers · 21 following

Follow

p/q2-q4!

Responses (1)

Write a response

What are your thoughts?

Rati
Jul 22

<3

Reply

More from Avtandil Ushikishvili

Avtandil Ushikishvili

How Instagram Solved the “Justin Bieber Problem” Using PostgreS...

“When we built Instagram, we never anticipated the scale of engagement driven...

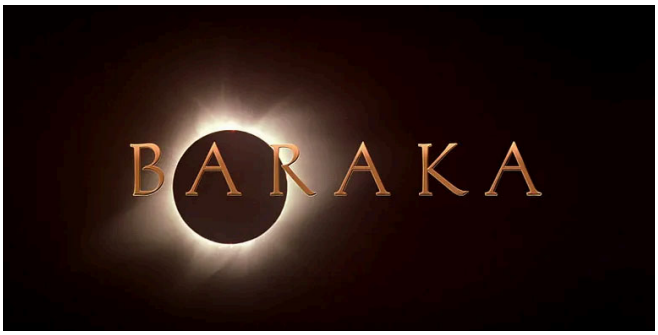
Jan 6 25

Avtandil Ushikishvili

How X-Men and Wolverine Saved Our Tests: Mutation Testing in .NET

Your tests seem invincible, but a tiny change breaks everything. Enter mutation testing — ...

Feb 10 16 1



Avtandil Ushikishvili

Object Cloning Distilled A.K.A Correct way to work with Prototyp...

Before deciphering the prototype pattern, I wondered how an exact copy of the object...

Dec 10, 2021 1

Avtandil Ushikishvili

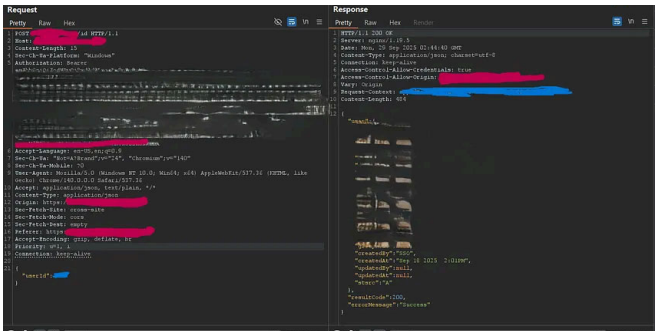
ვიზუალურად ესთეტიური ფილმების ნაბორი

თანამედროვე ფილმების უმეტესი ნაწილი ორიენტირებულია კომპიუტერულ სპეც...

Oct 20, 2019 6 1

See all from Avtandil Ushikishvili

Recommended from Medium



In InfoSec Write-ups by coffinxp

My 5-Minute Workflow to Find Bugs on Any Website

A step-by-step guide to my most effective, shortcut methods for bug bounty hunting.

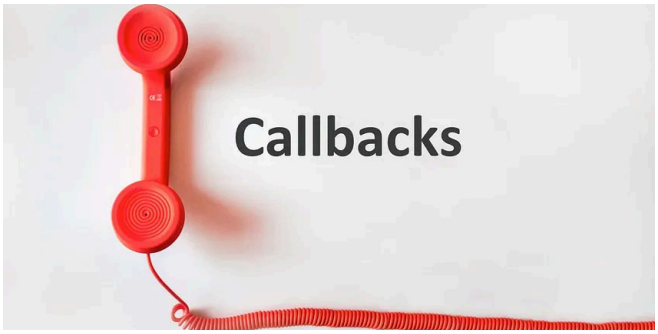
6d ago 782 5

Irsyad Muhammad Fawwaz

Authentication bypass via sequential user IDs in Microsoft...

If you're a penetration tester or bug bounty hunter, never skip SSO in your tests. It's one ...

4d ago 150 4



Oleh Slabak

Callback functions in C++

part#1

May 23 3 1

In Long. Sweet. Valuable. by Ossai Chinedum

I'll Instantly Know You Used Chat Gpt If I See This

Trust me you're not as slick as you think

May 16 24K 1452

Israel Aráoz Severiche

Hacking APIs: Tokens and Token Rotation

This article explains how attackers abuse refresh tokens when rotation or revocation is...

Facundo Fernandez

Top 10 Ways to Achieve Remote Code Execution (RCE) on Web...

Remote Code Execution (RCE) is a severe security vulnerability that allows attackers to...

Sep 24 Jul 1 114 1

See more recommendations